UNIVERSITAT POLITECNICA DE CATALUNYA

PARALLELISM

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

10th April 2019, Q2

Contents

1 Introduction

Talk about the Mandelbrot Set and about what we are going to do in this 3 sessions, in order.

2 Task decomposition analysis with *Tareador*

In this section, we had to analyse the two possible task granularities that could be exploited in the given program. To do it, we used the *Tareador* tool, which was very useful to see graphically the created tasks and which dependences are between them.

2.1 Point decomposition

In this decomposition strategy, a task corresponds with the computation of a single point (row, col) of the Mandelbrot set. Thus, we will have $row \times col$ tasks.

In order to analyse the potential parallelism of this strategy, we modified the code in *mandel-tar.c* to create several *Tareador* tasks. As we are using a point decomposition strategy, the tasks are created inside the most inner loop of the function mandelbrot. Figure 1 shows the fragment of the function that we modified.

```
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("point");

        ...

        tareador_end_task("point");
    }
}
```

Figure 1: Modified fragment of the *mandel-tar.c* code.

Once we did this, we executed interactively mandeld-tar and mandel-tar using run-tareador.sh script. The first one is used for timing purposes and to check for the numerical validity of the output (-o option) whereas with the second one we can visualize the Mandelbrot set.

The script has defined inside the size of the image to compute (-w option). In this case, the value was 8 to generate a reasonable task graph in a reasonable execution time. Hence, as we said before, the number of tasks will be $8 \times 8 = 64$.

Figures 2 and 3 below show the two task decomposition graphs (TDG) of the two possible executions. There are some nodes bigger than the others. This is because these nodes have executed more instructions than the rest. Therefore, these nodes represent pixels in white[1].

[1]The bigger the node, the more iterations it does in the do while fragment of the function. This make that the computed color is white.

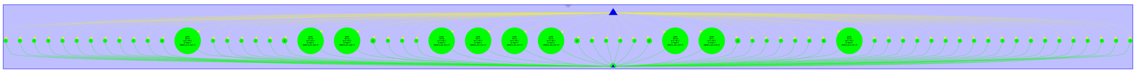

Figure 2: Mandel-tar task decomposition graph using the point decomposition strategy.

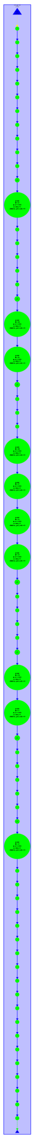

Figure 3: Mandeld-tar task decomposition graph using the point decomposition strategy.

This strategy will have more overhead of creation and termination of tasks than in the row strategy because it has to create more tasks. However, as a positive point the tasks are better distributed because if in a row there are many white areas, this work will not only be done by a single thread. Thus, it may end the execution earlier than in the row decomposition strategy.

2.2 Row decomposition

In this other strategy, a task corresponds with the computation of a whole row of the Mandelbrot set. This strategy only creates *row* tasks.

In this case, the code is not the same as before. We changed the creation of the *Tareador* tasks so that each time we enter a new row (second loop) a new task is created. The modified version of the code is shown below.

```
for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
        ...
    }
    tareador_end_task("row");
}
```

Figure 4: Modified fragment of the *mandel-tar.c* code.

Afterwards, we executed interactively mandel-tar and mandeld-tar again. We used the same size as before. Hence, the total number of tasks will be 8. The graphical results we got can be seen in figures 5 and 6.

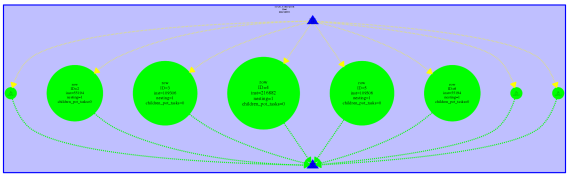

Figure 5: Mandel-tar task decomposition graph using the row decomposition strategy.

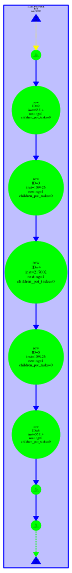

Figure 6: Mandeld-tar task decomposition graph using the row decomposition strategy.

In this strategy, the overhead time of creation and termination of tasks will be very small in comparison with the point strategy because it has to create less tasks. Nevertheless, it may happen that in a full row, all of its pixels must be white. In this case, it can happen that while other threads have finished their work, this other thread is still executing the row. Consequently, it is possible that the execution time may be bigger than in the point decomposition strategy.

2.3 Characteristics of the TDG

We saw in the previous sections that the execution of the mandel-tar has a very different task dependence graph than the execution of the mandeld-tar.

On the one hand, we can see that in mandel-tar every point is independent from the others. Consequently, we could parallelize that fragment of the code using OpenMP clauses.

On the other hand, in mandeld-tar we can see that all iterations have became sequential. In this situation, we do not gain anything by parallelizing the code, but we increase the execution time because of the overhead of creation and termination of tasks.

Using the *Tareador* we could see which variable was responsible of creating those dependences. We did the following: Right Click into a task -> Data View -> Edges-out -> Real Dependency. Figure 7 shows the result we got.

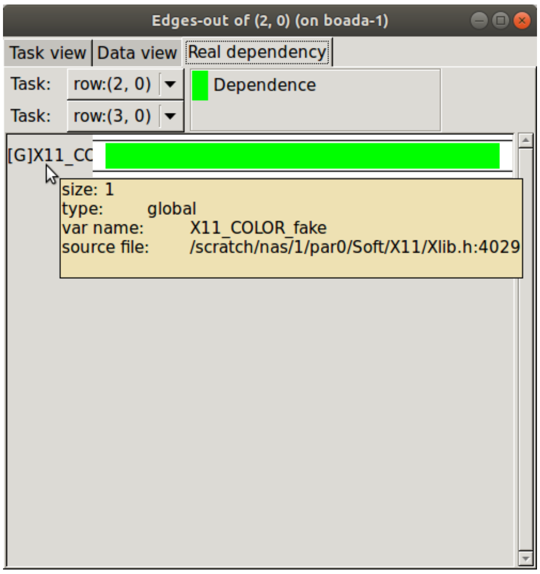

Figure 7: Variable that provoques the dependences between tasks.

We can see that there is only one variable that is causing all the dependences: X11_COLOR_fake. Observing the code, we noticed that the only difference between mandel-tar and mandeld-tar was a fragment of the code that was only executed in mode _DISPLAY_ (mandeld-tar):

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 8: Fragment of the *mandel-tar.c* code.

Therefore, variable X11_COLOR_fake is used at least in one of the functions XSetForeground and XDrawPoint. We could protect this section of code in the parallel OpenMP code using the *critical* clause to define a region of mutual exclusion where only one thread can be working at the same time.

```
#if _DISPLAY_
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;

    #pragma omp critical
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
#else
    output[row][col]=k;
#endif
```

Figure 9: Fragment of the *mandel-tar.c* code using the *critical* clause to protect a fragment of the code.

FALTA AIXO: Reason when each strategy/granularity should be used.

You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelization strategies that you have done. Include both the PDF and source codes in a single compressed tar file (GZ or ZIP). Only one file has to be submitted per group through the Raco website.

3 Point decomposition in OpenMP

In this section, we are going to explore different options in the OpenMP tasking model to express the Point decomposition for the Mandelbrot computation program. We will analyse the scalability and behaviour of these options.

3.1 OpenMP Task Implementation

The aim of this tasking model strategy is to create a task for each point. Figure 10 show the OpenMP clauses we used to implement the task strategy (the code can be found in file *mandel-omp-task-point.c* in codes folder).

```
for (int row = 0; row < height; ++row) {
    #pragma omp parallel
    #pragma omp single
    for (int col = 0; col < width; ++col) {
         #pragma omp task firstprivate(col)
         {
             ...
             #pragma omp critical
             if (setup_return == EXIT_SUCCESS) {
                  XSetForeground (display, gc, color);
                  XDrawPoint (display, win, gc, col, row);
             }
             ...
         }
    }
}
```

Figure 10: Fragment of the *mandel-omp-task-point.c* code showing the OpenMP clauses to implement the task strategy.

For every row, only one thread is creating all the tasks and inserting them into a pool of tasks, while the other threads are taking tasks from the pool and executing them. Nevertheless, the "role" of creating tasks changes for every row. Thus, each iteration of the col loop will be executed as an independent task. The firstprivate clause is used in order to avoid problems of data races[2]. Finally, the critical clause is used to honour the dependences we detected for the graphical version in the previous section. However, this section of the code will not be called since we will not execute the graphical version (only mandel-tar).

The following figure shows the execution flow of the program using the *Paraver* tool.

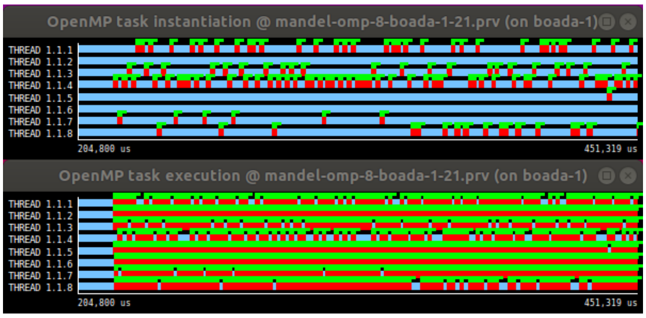

Figure 11: Zoomed part of the execution flow using the task strategy.

[2]We have not used the private clause because it initializes the variable with a random value, and we wanted the real value of the variable (iteration).

We can observe the effect of the single clause. In the upper part (task instantiation) only one thread is creating the tasks and inserting them in the task pool. In the bottom part (task execution) we can see that the other threads execute the tasks as they are put in the pool. The role of the task creator changes every time we finish a row.

In total, there are 640000 tasks created and executed. This is because the image has size 800×800. We have seen this using the "OMP_parallel_functions.cfg" and "OMP_state_profile.cfg" configurations. Figure 12 shows the results obtained.

Task profile (execution and instantiation) @ mandel-omp-8-boada-1-21.prv (on b...

	Unknown type 60,000,023	Unknown type 60,000,024
THREAD 1.1.1	87,442	188,800
THREAD 1.1.2	74,776	7,200
THREAD 1.1.3	83,265	112,800
THREAD 1.1.4	82,858	208,800
THREAD 1.1.5	76,021	7,200
THREAD 1.1.6	75,429	4,000
THREAD 1.1.7	82,797	10,400
THREAD 1.1.8	77,412	100,800
Total	640,000	640,000
Average	80,000	80,000
Maximum	87,442	208,800
Minimum	74,776	4,000
StDev	4,369.63	79,964.99
Avg/Max	0.91	0.38

Figure 12: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

The parallel construct is executed 800 times (number of rows) and the single worksharing construct is executed 800 times for each thread. Thus, the single clause is executed 6400 times ($rows \times threads$). All threads will execute the single line but only one will "gain". This thread will be responsible for creating the tasks of that row.

Finally, figures 13 and 14 show the time and speedup plots.

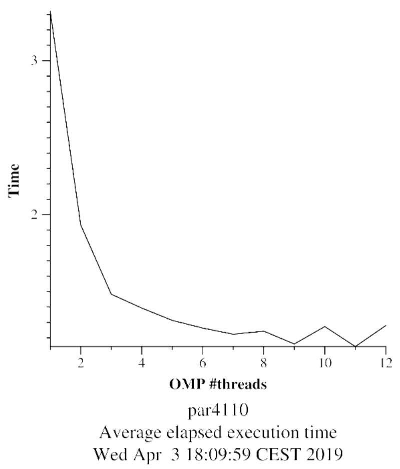

Figure 13: Execution time plot varying the number of threads.

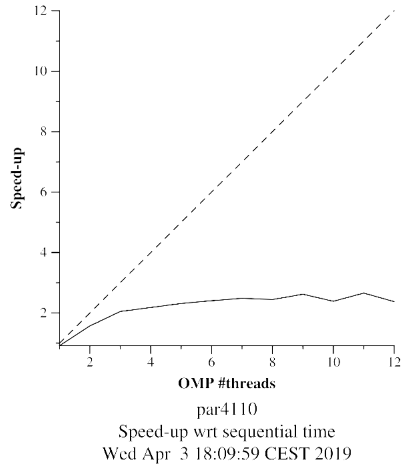

Figure 14: Speedup plot varying the number of threads.

In the first plot, the execution time is reduces as we increase the number of threads until the time stays constant. However, we think that if we increase a lot the number of threads, the execution time would end up increasing because of the overheads.

On the other hand, in the second plot the speedup increases a lot with 2-4 threads but ends up constant as we increase the number of threads.

REPASSAR EL SEGUENT: The scalability is not appropiate because the speedup stays at 2 (reduction of time of T_1) even though we have 8 threads.

3.1.1 OpenMP Taskwait Variant

In this variant of the previous code, only one thread (the one that gets access to the single region), traverses all iterations of the row and col loops, generating a task for each iteration of the innermost loop (point). To do it, we have introduced "#pragma omp taskwait" at the end of each iteration of a row. Consequently, the creator thread must wait until all tasks for a row finish. After that, the thread will advace one iteration of the row loop and generate a new bunch of tasks. This new version of the code can be found in *mandel-omp-task-point-taskwait.c*.

```c
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
            ...
        }
    }
    #pragma omp taskwait // waiting point for all child tasks
}
```

Figure 15: Fragment of the *mandel-omp-task-point-taskwait.c* code showing the OpenMP clauses to implement the task strategy with the taskwait variant.

We can clearly see in figures 16, 17 and 18 that the only thread that is creating tasks is the thread 0. Moreover, in the zoomed figure we can observe that every time it creates some tasks, then it has to wait a little bit until it can create tasks again. This is because of the taskwait clause we added at the end of the outermost loop. Thread 0 will create all the tasks for a unique row and will wait until all these tasks terminate.

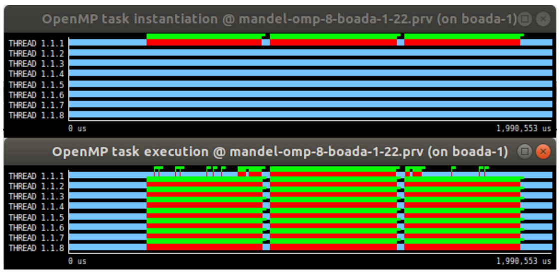

Figure 16: Execution flow using the taskwait strategy.

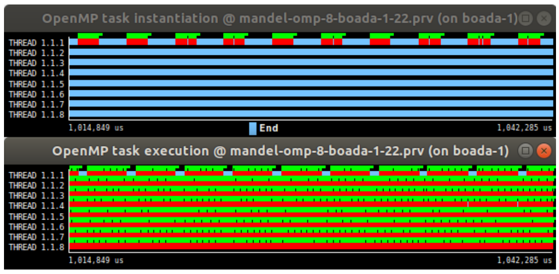

Figure 17: Zoomed part of Figure 16.

The number of created and executed tasks is still 640000 as we have not modified the size of the image. Nevertheless, the number of calls to parallel is 1, and the number of calls to the single worksharing construct is now 8, the number of threads. Besides, the taskwait clause will be executed 800 times (number of rows). The granularity is the same than before. Each task has exactly one iteration of the innermost for loop.

Task profile (execution and instantiation) @ mandel-omp-8-boada-1-22.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89,224	640,000
THREAD 1.1.2	70,094	-
THREAD 1.1.3	79,978	-
THREAD 1.1.4	84,120	-
THREAD 1.1.5	73,685	-
THREAD 1.1.6	74,480	-
THREAD 1.1.7	84,387	-
THREAD 1.1.8	84,032	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	89,224	640,000
Minimum	70,094	640,000
StDev	6,185.05	0
Avg/Max	0.90	1

Figure 18: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

Figures 19 and 20 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the previous section.

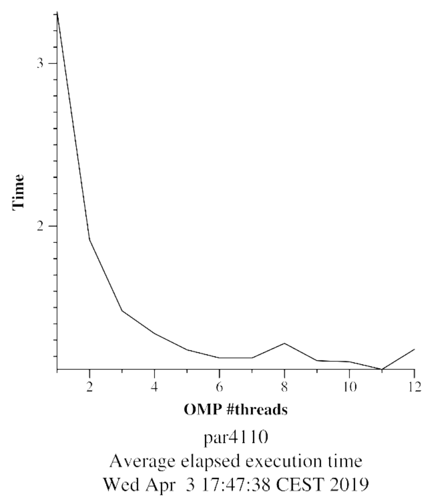

Figure 19: Execution time plot varying the number of threads.

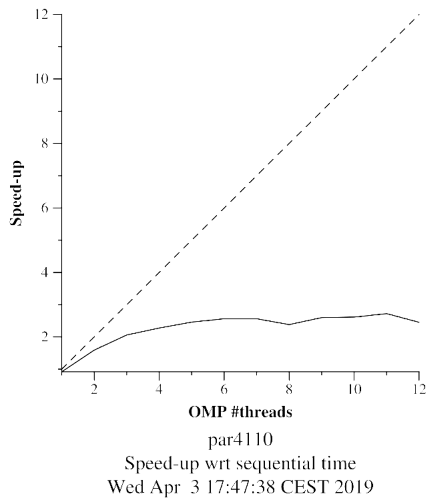

Figure 20: Speedup plot varying the number of threads.

3.1.1.1 The Taskwait Clause

In reality, the taskwait clause is not necessary because the tasks do not have any dependence between them. We saw in section 2 that the mandel-tar binary does not produce any kind of dependence between tasks. Thus, we can create and execute all tasks without having to wait until some of them terminate. The modified version of the code can be found in the file *mandel-omp-task-point-taskwait-without.c*.

In order to see the differences with respect to the version with the taskwait clause, we have repeated the evaluation of scalability and tracing. The results are shown below.

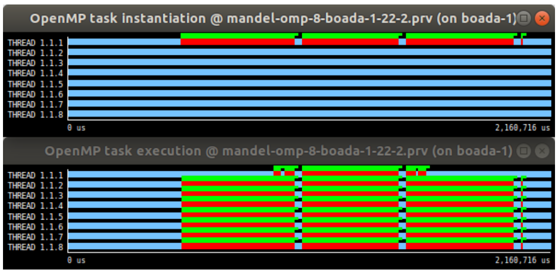

Figure 21: Execution flow using the taskwait strategy without the barrier.

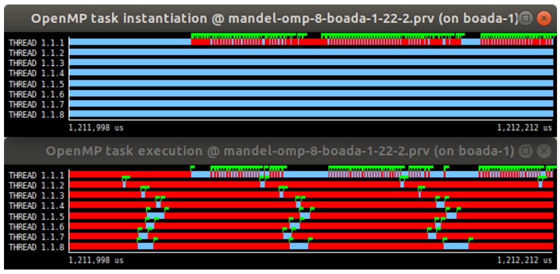

Figure 22: Zoomed part of Figure 21.

The number of tasks created/executed has not changed because we have not touched the task clause or the parallel and single clauses. However, as we have deleted the taskwait clause, now the creator thread (thread 0 in our case) does not have to wait until the tasks of the same row terminate. Consequently, it can create all possible tasks of the program at the same time. Nevertheless, we see in figures 21 and 22 that this is not happening as expected.

We though that thread 0 would create all possible tasks and then execute the remaining tasks inside the task pool. What is happening is that this thread creates some tasks, then stops the task generation, then executes some taks, and then proceeds generating new tasks. After reasoning about this, we have concluded that the task pool that OpenMP has is limited, it has a maximum number of threads inside the pool. As a consequence, when thread 0 reaches this limit it has to stop creating tasks, so it begins to execute the created tasks until the total number of tasks inside the pool is reduced. Afterwards, it begins to create tasks again.

On the other hand, we can see that the scalability plots (time and speedup) have improved a bit even though it is not a significative change.

CANVIAR AQUESTES GRAFIQUES

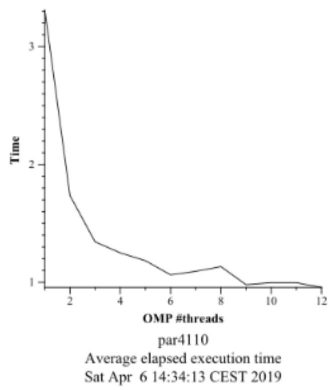

Figure 23: Execution time plot varying the number of threads.

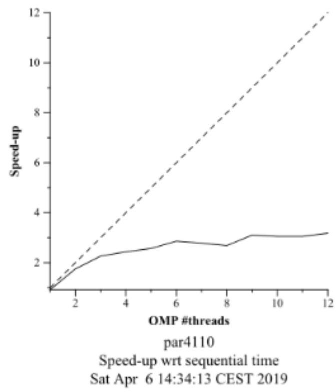

Figure 24: Speedup plot varying the number of threads.

To sum up, we can delete the taskwait clause because it is not necessary for mandel-tar. This change improves a little bit the performance of the execution. Moreover, without this clause the creator thread has to stop generating tasks because it reaches the maximum limit of the pool. Thus, it will be changing from creating to executing and vice versa very often.

3.1.2 OpenMP Taskgroup Variant

In this other variant, we define a region in the program where the thread will wait for the termination of all descendant (not only child) tasks. Figure 25 shows a section of the modified code that can be found in file *mandel-omp-task-point-taskgroup.c* inside the codes directory. Now, the tasks of the same row are generated without any specific order. However, the creator thread has to wait until all the tasks of the same row terminate.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskgroup
    {
        for (int col = 0; col < width; ++col) {
            #pragma omp task firstprivate(row, col)
            {
                ...
            }
        }
    }
}
```

Figure 25: Fragment of the *mandel-omp-task-point-taskgroup.c* code showing the OpenMP clauses to implement the task strategy with the taskgroup variant.

In the execution flow we obtained with *Paraver* we can see that there is no differenece at all with respect to the execution flow of the previous section. This is because in reality the taskgroup clause does not have descendants of its childs. As a consequence, it does the same effect than using the taskwait clause.

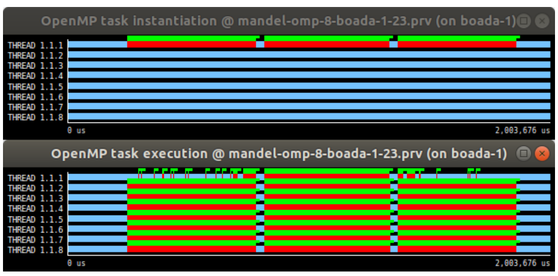

Figure 26: Execution flow using the taskwait strategy without the barrier.

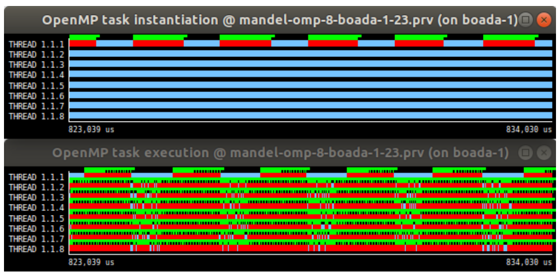

Figure 27: Zoomed part of Figure 26.

The number of created tasks is the same (640000). Moreover, the number of calls to parallel and the single worksharing contruct is the same (1 and 8 respectively). The only difference is that in this variant the taskgroup clause is executed 800 times (number of rows).

Task profile (execution and instantiation) @ mandel-omp-8-boada-1-23.prv (on boada-1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,180	640,000
THREAD 1.1.2	79,923	-
THREAD 1.1.3	78,184	-
THREAD 1.1.4	76,028	-
THREAD 1.1.5	81,165	-
THREAD 1.1.6	80,066	-
THREAD 1.1.7	75,837	-
THREAD 1.1.8	80,617	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,180	640,000
Minimum	75,837	640,000
StDev	3,623.98	0
Avg/Max	0.91	1

Figure 28: Table with the number of executed tasks in the left column and the number of created tasks in the right column.

Figures 29 and 30 show again the time and speedup plots. We have not noticed any significant difference between these plots and the plots of the two previous sections.

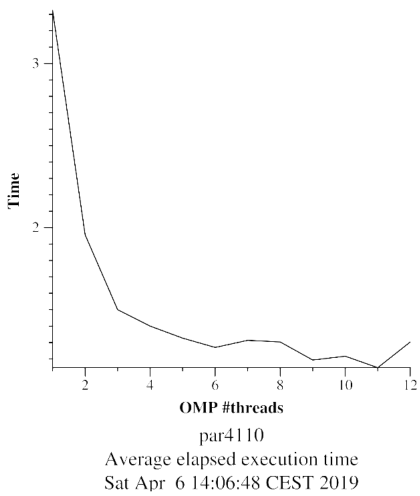

Figure 29: Execution time plot varying the number of threads.

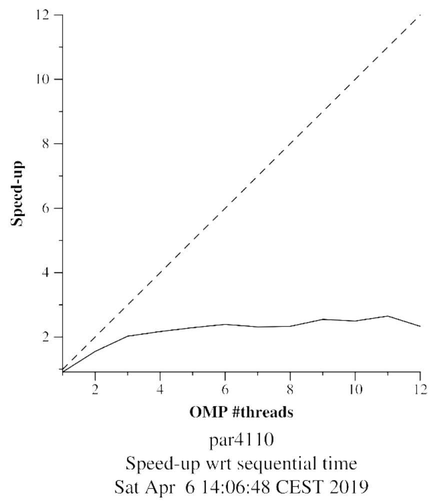

Figure 30: Speedup plot varying the number of threads.

3.2 OpenMP Taskloop Implementation

In this new strategy, the taskloop clause is used. It generates tasks out of its iterations of a for loop, allowing to better control the number of tasks generated or their granularity (num_tasks or grainsize). This version of the code can be found in the file *mandel-omp-task-point-taskloop-800*.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(800)
    for (int col = 0; col < width; ++col) {
        ...
    }
}
```

Figure 31: Fragment of the *mandel-omp-task-point-taskloop-800.c* code showing the OpenMP clauses to implement the task strategy with the taskloop variant.

If we create 800 tasks of the mostinner loop, in reality we are executing an equivalent version of the second task version we analysed. In order to analyse the performance of this strategy we will use the trace generated with *Paraver* and using the appropiate configuration files.

FALTEN ELS TRACES, CONFIGURACIONS I DIR PERQUE TE MILLOR RENDIMENT QUE LA VERSIO 22

3.2.1 The Nogroup Clause

The taskloop construct has an implicit task barrier (taskgroup) at the end. This is causing the effect of a taskwait clause. As we reasoned before about that we could delete the taskwait clause because there are no dependences between tasks, we can do the same with the tasgroup clause. To do it, the taskloop construct accepts a nogroup clause that eliminates this implicit barrier.

Now, we will explore how this new version behaves in terms of performance when using 8 threads and for different task granularities (800, 400, 200, 100, 50, 25, 10, 5, 2 and 1 tasks). All versions of the codes can be found in files *mandel-omp-task-point-taskloop-{granularity}-nogroup*.

FALTA EL PLOT. EN ROGER SAP COM FER-LO BE. TAMBE POSO ELS STRONG PLOTS?

As we can see in Figure XXXXXXXXXXXXXXXXXXX, the bigger the granularity, the bigger the execution time (and the lower speedup). The main reason is that we have to create more tasks. Thus, the overhead of creation and termination of tasks is bigger. The best option using this strategy is small values of granularity. In this situation, only 1 or 2 threads will execute the full row, while the other ones can execute the following rows.

4 Row decomposition in OpenMP

In this section, we are going to analyse the scalability and tracing of the Row decomposition strategy using the best implementation we got with the Point strategy. Afterwards, we will compare both results.

The best implementation in the previous section was using the taskloop clause. Thus, the new code is the following:

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop firstprivate(row) num_tasks(800) nogroup
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        ...
    }
}
```

Figure 32: Fragment of the *mandel-omp-task-row-taskloop-800-nogroup.c* code showing the OpenMP clauses to implement the task strategy with the taskloop variant.

5 Conclusion

6 Annex

6.1 Blablabla

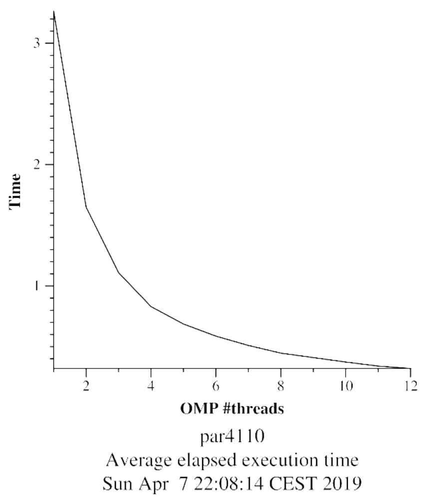

Figure 33: Execution time plot varying the number of threads (Num. tasks = 1).

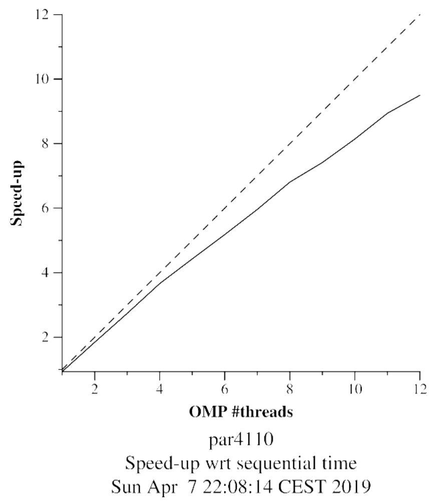

Figure 34: Speedup plot varying the number of threads (Num. tasks = 1).

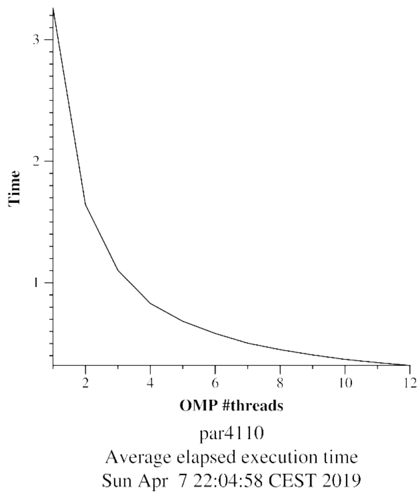

Figure 35: Execution time plot varying the number of threads (Num. tasks = 2).

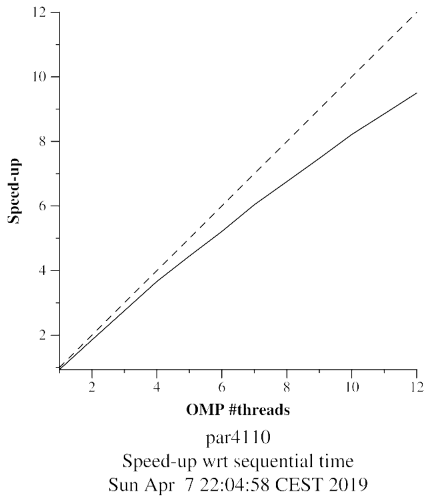

Figure 36: Speedup plot varying the number of threads (Num. tasks = 2).

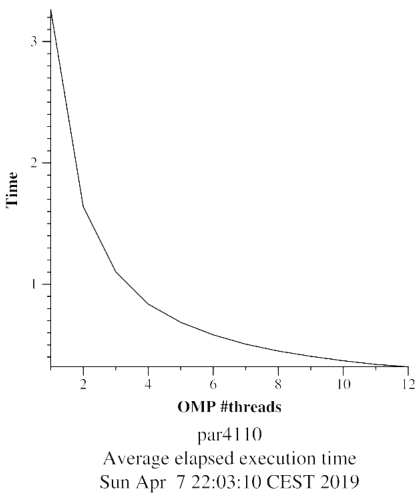

Figure 37: Execution time plot varying the number of threads (Num. tasks = 5).

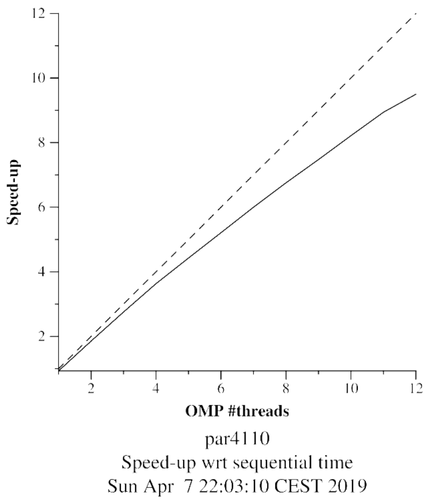

Figure 38: Speedup plot varying the number of threads (Num. tasks = 5).

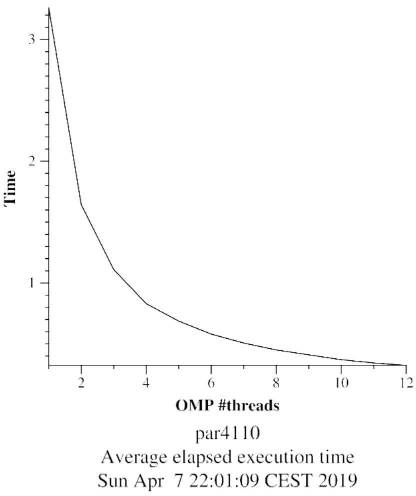

Figure 39: Execution time plot varying the number of threads (Num. tasks = 10).

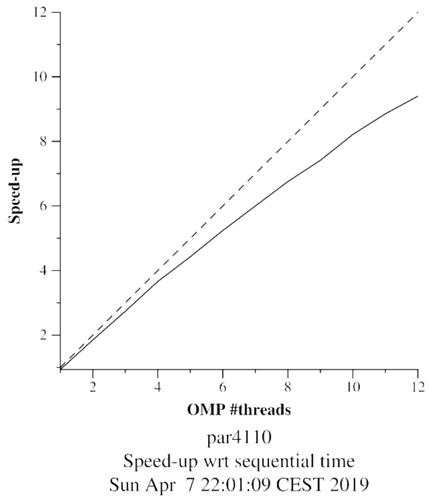

Figure 40: Speedup plot varying the number of threads (Num. tasks = 10).

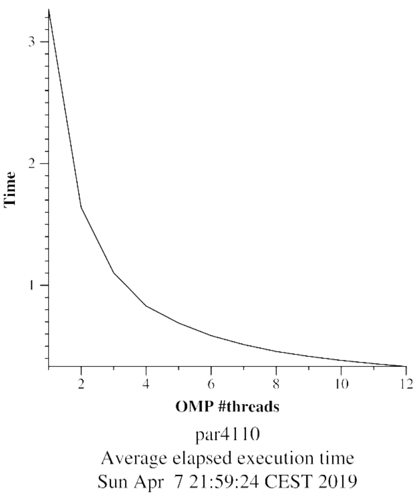

Figure 41: Execution time plot varying the number of threads (Num. tasks = 25).

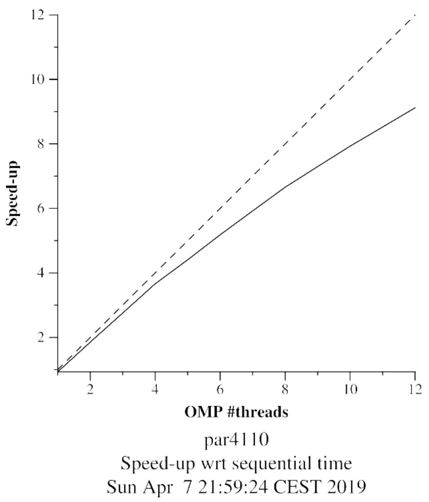

Figure 42: Speedup plot varying the number of threads (Num. tasks = 25).

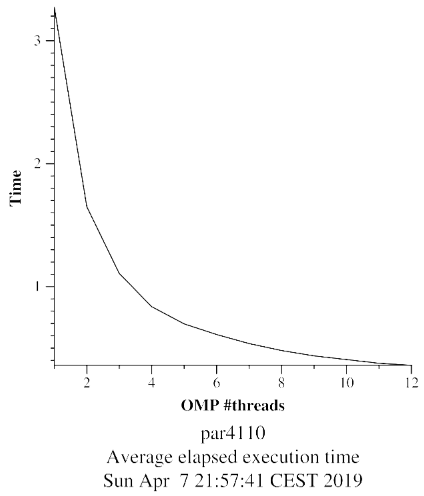

Figure 43: Execution time plot varying the number of threads (Num. tasks = 50).

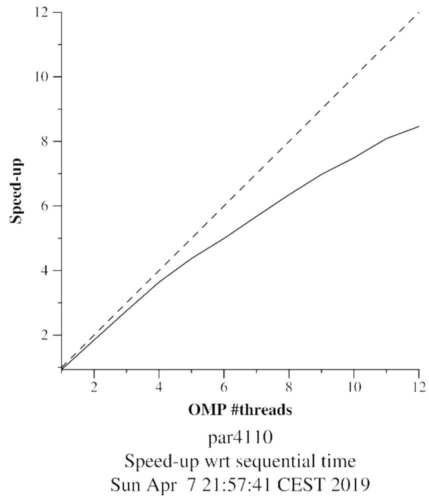

Figure 44: Speedup plot varying the number of threads (Num. tasks = 50).

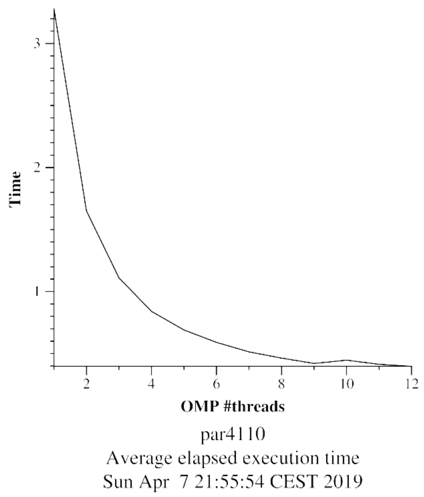

Figure 45: Execution time plot varying the number of threads (Num. tasks = 100).

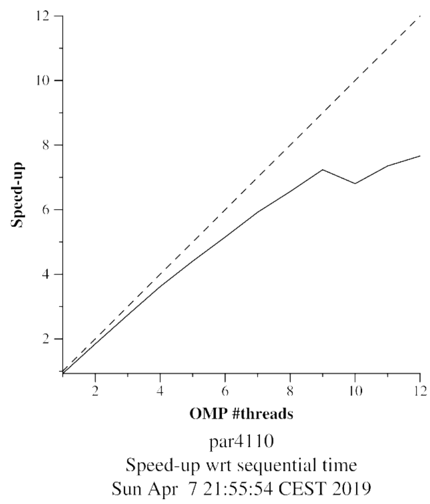

Figure 46: Speedup plot varying the number of threads (Num. tasks = 100).

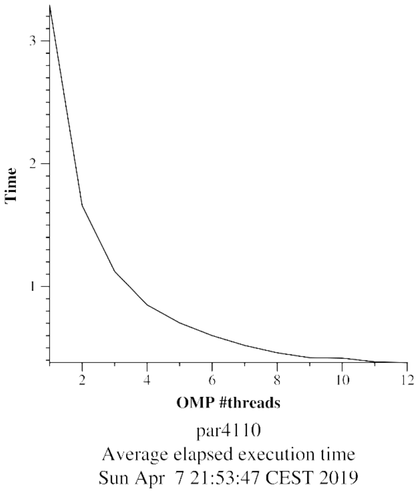

Figure 47: Execution time plot varying the number of threads (Num. tasks = 200).

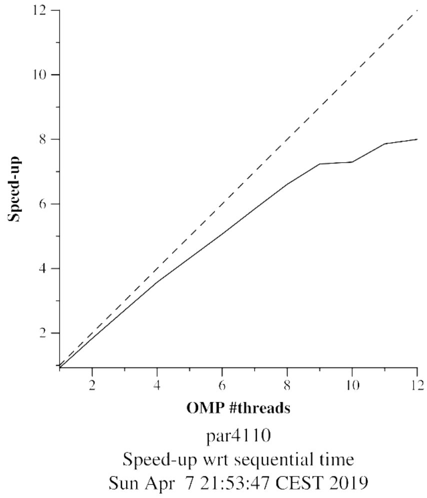

Figure 48: Speedup plot varying the number of threads (Num. tasks = 200).

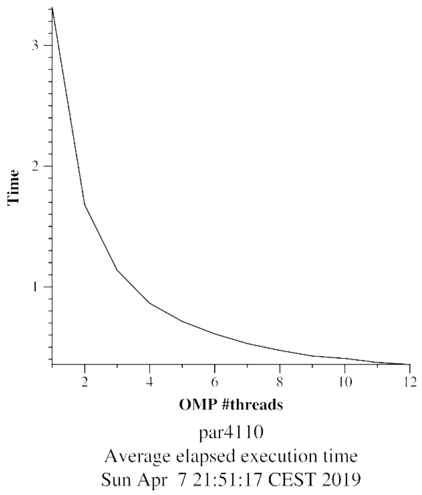

Figure 49: Execution time plot varying the number of threads (Num. tasks = 400).

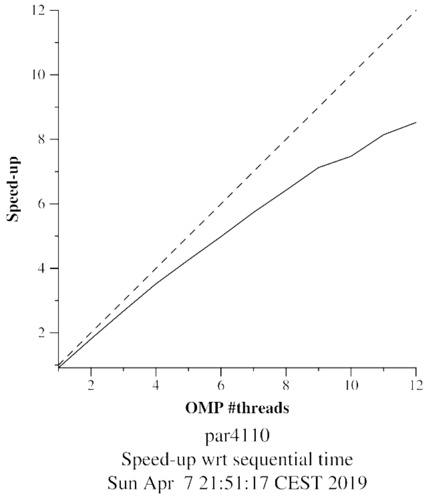

Figure 50: Speedup plot varying the number of threads (Num. tasks = 400).

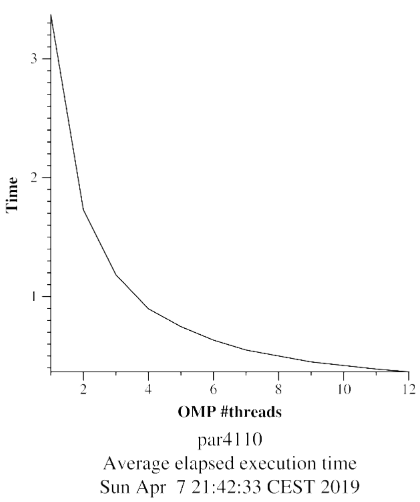

Figure 51: Execution time plot varying the number of threads (Num. tasks = 800).

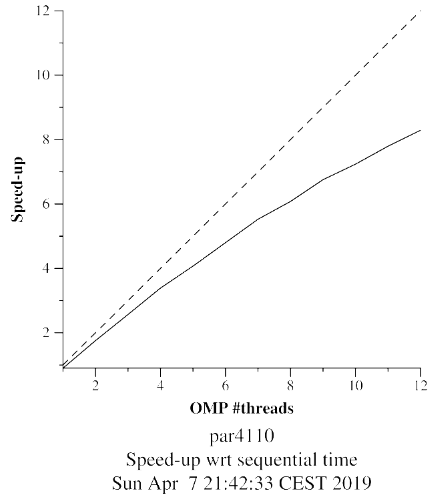

Figure 52: Speedup plot varying the number of threads (Num. tasks = 800).

6.2 Blablabla2

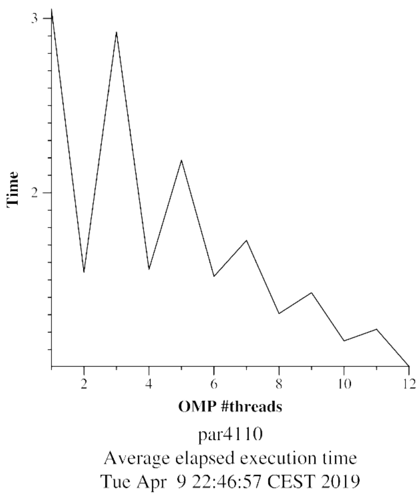

Figure 53: Execution time plot varying the number of threads (schedule(static)).

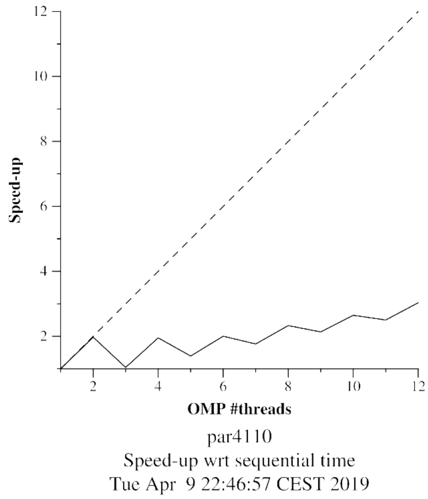

Figure 54: Speedup plot varying the number of threads (schedule(static)).

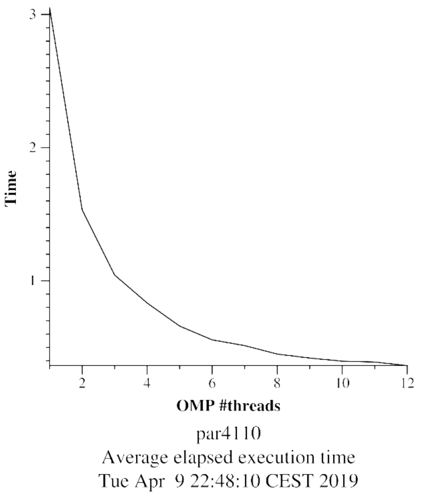

Figure 55: Execution time plot varying the number of threads (schedule(static, 10)).

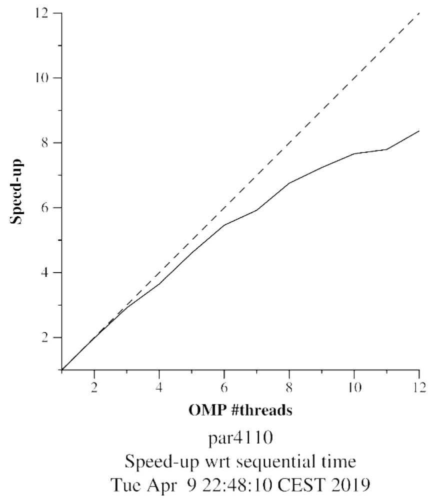

Figure 56: Speedup plot varying the number of threads (schedule(static, 10)).

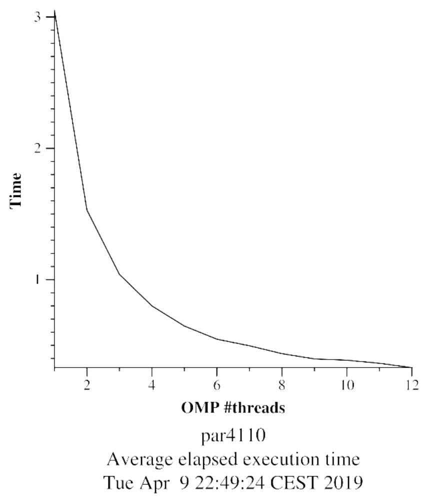

Figure 57: Execution time plot varying the number of threads (schedule(dynamic, 10)).

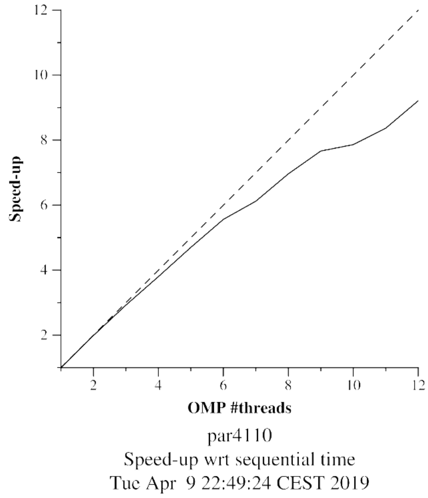

Figure 58: Speedup plot varying the number of threads (schedule(dynamic, 10)).

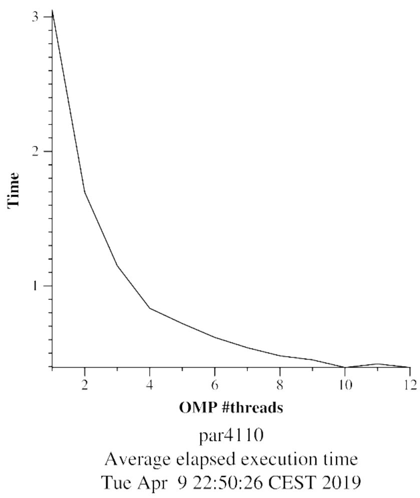

Figure 59: Execution time plot varying the number of threads (schedule(guided, 10)).

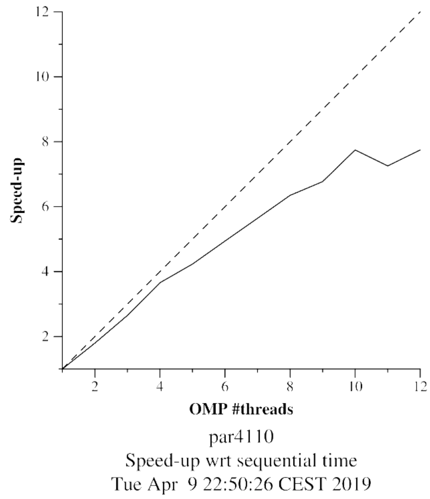

Figure 60: Speedup plot varying the number of threads (schedule(guided, 10)).